Data compression algorithms for flow tables in Network Processor RuNPU.

Nikita Nikiforov
Lomonosov Moscow State University
Moscow, Russia
nickiforov.nik@gmail.com

Dmitry Volkanov
Lomonosov Moscow State University
Moscow, Russia
volkanov@asvk.cs.msu.ru

***Abstract*—This paper addresses the problem of packet classification within a network processor (NP) architecture without the separate associative device. By the classification, we mean the process of identifying a packet by its header.The classification stage requires the implementation of data structures to store the flow tables. In our work we consider the NP without the associative memory. Flow tables are represented by an assembly language program in the NP. For translating flow tables into assembly language programs, a tables translator was used. The main reason for implementing data compression algorithms in a flow tables translator is that modern flow tables can take up to tens of megabytes. In this paper we describe the following data compression algorithms: Optimal rule caching, recursive endpoint cutting and common data compression algorithms. An evaluation of the implemented data compression algorithms was performed on a simulation model of the NP.**

***Index Terms*—Network processor, software-defined networks, packet classification, data compression.**

I. Introduction

At present, software-defined networks (SDN) are in active development and require high-performance switches [1]. The main functional element of the high-performance SDN switch is a programmable network processor (NP). The network processor is a system-on-chip specialized for network packet processing. In this work we consider a programmable NP. A programmable NP is one that supports on-the-fly modification of the packet processing program and the set of header fields to be processed.

In this article we discuss data compression algorithms used for flow tables. Flow tables are needed for packet classification process. A flow table is the set of rules defined by OpenFlow protocol. OpenFlow is one of the most common protocols for controlling a network SDN switch. This paper considers OpenFlow version 1.3 [2]. Each rule contains a match field, a bit string by witch a packet can be identified and a set of actions, that the NP performs on this packet. Classification is the process of the identification of a network packet by its header.

This article has the following structure: in second section we introduce problem, in third section we introduce the NP architecture and flow tables translator, in fourth section we describe related work, in fifth section we describe data compression algorithm implementation and in sixth section we introduce our evaluation methodology.

II. The problem

Let us consider OpenFlow tables formalisation. An ordered set of all considered attributes is denoted as $I = \{m_1, m_2, \ldots, m_k\}$. Every attribute m_i from the set I is described by a bit string $m_i \in \{0, 1, *\}_i^W$. In this article symbol $*$ denotes any bit. But, if $\exists m_i^j \in m_i$ and $m_i^j = *$, then for $\forall m_i^k$, where $k > j$, $m_i^k = *$. The length of the attribute is denoted $len(m_i) = W_i$.

Flow tables are represented by a set of rules $R = \{r_1, r_2, \ldots, r_n\}$. With every rule r_i binding the features:

- An index i;
- A priority $p_i \in Z_+$;
- A vector of values of attributes $f_i = \{f_i^1, f_i^2, \ldots, f_i^k\}$, where f_i^j is an attribute value $m_j \in I$.
- A set of actions $A_i = \{a_1, a_2, \ldots, a_z\}$.

A network packet header x and its metadata with vector values of attributes $g = \{g^1, g^2, \ldots, g^k\}$ ($x \to g$), a match rule $r_i \in R$ with a vector of values of the attributes $f_i = \{f_i^1, f_i^2, \ldots, f_i^k\}$ and a priority p_i (a rule $r_i \in R$ identifies a network packer with a vector values of attributes g), if:

1) a vector values of attributes g match a vector of values of the attributes f_i, $\forall g_i \in g$, $len(g_i) = len(f_i)$. $\forall f_i^{lj} \in f_i^l$, $f_i^{lj} \in \{*, g^{lj}\}$, $l = \overline{1,k}$;
2) a priority p_i is the highest among all rules $r_j \in R$, if a vector g match a vectorf_j.

The set of rules R must satisfy the following constraint. For any two rules $r_i, r_j \in R, r_i \neq r_j$, if their vectors of values intersect, there is a set of attribute values. This set corresponds to vectors of values of attributes of both rules $p_i \neq p_j$.

Let us introduce the function for network packet identification $x \to g$in flow table R, (denotes as $R(x)$). It returns a set of actions, that corresponded to the rule $x \to g$.

$R(x) = A_{r_i}$, where A_{r_i} is the set of actions $r_i \in R$.

We need to introduce a **similar concept** of the sets of rules R_1 and R_2. The set R_1 is similar to the set R_2, when for any network packet header, that can be identified by some rule from the set $r_i \in R_1$, and there exists another rule that identifies it as $r_j \in R_2$, and $A_i = A_j$.

We need to develop an algorithm for compressing flow tables. This algorithm must translate an input flow table (a set of rules R_1) into a new compressed set of rules R_2.

1) The set of rules R_1 is similar to the set of rules R_2.
2) The cardinality of the set R_2 must be lower than the cardinality of the set R_1.

III. Network processor architecture

In the considered NP the pipeline architecture is used, with each pipeline consisting of 10 computing blocks. To avoid complex memory organization, there is no associative memory in the considered NP. The NP uses the same memory both for commands and data.

Let us consider the pipeline NP architecture. Each computing block has an access to the memory area where the program with data is located. There is a limit of 25 clock cycles per packet on each processing block. There is up to 512 kilobytes to store assembly language program representing flow tables. Due to the instruction set architecture, there is no separate memory area where data is stored. Therefore, the microcode contains all the data, required to classify packets.

A. Flow tables translator

Flow tables translator is a tool that is executed on CPU. It is used for flow tables translating into assembly language programs, that can be interpreted by NP. Flow tables translator uses tree structures for flow table representation. Every node of the tree structure can be associated with a table rule. After building a tree every node is translated into a part of an assembly language program. Here is a flow tables translator workflow:

1) Load a flow table from file.
2) Check every rule in the table.
3) Build a tree structure from a set of rules.
4) Translate every node into a part of an assembly language program.
5) Combine all translated parts into the one assembly language program.
6) Add a header that corresponds to used protocol.
7) Write the assembly language program into file.

This tool was implemented in work [3].

IV. Related work

In this section we introduce a review on data compression algorithms, that already used for other network processors [4]. To choose algorithms for implementation in NP we used the following criteria:

1) Compression rate, is needed for algorithm performance evaluation.
2) Evaluation of compression algorithm complexity.
3) Usability of compressed flow tables without decompression.
4) The necessity to use external memory by the algorithm.

A. Most common data compression algorithms

Data compression algorithms have evolved over the years. Nowadays compression algorithms can be used in many different ways. In this section we describe the algorithms that compress data in binary format. There are most known of them: Huffman codding, JPEG, LWZ, zip. These algorithms require decompression for data usage. And this is why we will not use them in our flow table translator.

B. Optimal rule caching

Optimal rule caching algorithm is more specific data compression algorithm. It is used for table compressing in SND switches [5]. It is based on search tree structure, that is built based on rules usage frequencies. There are two trees: the first tree consist from most used rules. This tree is translated into assembly language program. The second tree consist from other rules, it is stored in CPU memory.

C. Recursive end point cutting

Recursive end-point cutting algorithm is based on HyperSplit tree usage. Compressing is performed by destroying duplication rules [6]. This algorithm permits operations with flow tables without full rebuilding tree. By rules duplication we understand the following rules:

- A rule, storing in node duplicates of the rules in leaf nodes. (particle duplication).
- A rule, storing in node duplicates of the rules in all leafs nodes. (full duplicating rule).

This algorithm recursively uses NewHypersplit tree to remove duplicate rules from the currently being built tree. The deleted duplicate rules are then collected into a second rule table, called a recursive table, to build a second tree. It is possible that duplicate rules still exist in the second tree, and some of them are also removed and used to build the third tree.

This tree building process is performed recursively while there are duplicate rules in the last tree.

D. Algorithms comparison

Let us describe data compression algorithms comparison in table I. Each algorithm has its own pros and cons.

1) **Optimal rule caching** — has the highest compression ratio, and is quickly implemented in the considered NP. The need to use external memory imposes additional overhead on some packet processing.
2) **Recursive end-point cutting** — has the lowest compression ratio, it is more difficult to implement than the optimal rule caching algorithm. Moreover,this algorithm does not require the use of external memory.
3) **Common data compression algorithms** — have good compression ratios on average, but require data decompression.

TABLE I: Data compression algorithms comparison

Name	complexity of construction	Compression rate	CPU memory usage	Decompression needed
Optimal rule caching	$O(N^2)$	$0.1 \dots 0.9$	yes	no
Recursive end-point cutting	$O(N * log(N))$	0.1	no	no
Common data compression algorithms	$O(K * \log_2 N)$	$0.1 \dots 0.8$	no	yes

V. Our solution

In this section we introduce our solution of flow tables compressing.

A. Flow table optimization

First of all we need to introduce operation **getting last important bit** $last(m_i) = j$, $m_i^j \in \{0, 1\}$ and $m_i^{(j+1)} = *$. We claim that the rules $r_i \in R$ and $r_j \in R$ are the **same**, if $\forall u \in len(f_i)\ last(f_i^u) = last(f_j^u) = l$, but $f_i^{ul} \neq f_j^{ul}$ and $A_i = A_j$. For flow table optimization we need to remove all **same** rules.

B. Main flow table compression algorithm

Let us introduce a packet header distribution P, where p_x mean network packet income probability$x \to g = \{g^1, g^2, \dots, g^k\}$. We need a correction ratio $T_P(R_1, R_2)$, where R_1 and R_2 are two different flow tables. Thus correction ratio means probability of incoming network packet header by distribution P. As well as probability of identifying this network packet by rules $r_1 \in R_1$ and $r_2 \in R_2$. Moreover the sets of actions of this rules are similar $A_1 = A_2, A_1 \in r_1, A_2 \in r_2$.

$$T_P(R_1, R_2) = \sum_{x \to g, R_1(x) = R_2(x)} p_x$$

The optimal correction ratio for flow table R and a number of rules n and a network packet header distribution P is:

$$\zeta(n, R, P) = \max_{R_i, |R_i| <= n} T_P(R, R_i)$$

Let p^i be probability of choosing rule $r_i \in R$, in distribution P. Let rules in flow table R be in not-increasing order of their probabilities. Then:

$$\zeta(n, R, P) \geq \sum_{i \in [1,n]} p^i + 1 - \sum_{i \in [1,n_0]} p^i \geq n/n_0$$

This algorithm needs exploration and building a flow table R_a, based on input flow table R. There is a minimal set of rules (n_0) and a maximum optimal correction ratio $\zeta(n, R, P)$.

C. Software solution

In this section we introduce software workflow of our algorithm. First of all we need to add a new fields in tree structure nodes for our algorithm.

- A probability into tree node. It must be filled if node contains rule.
- A sum of probabilities of leaf nodes.

Let us introduce program operation for split tree.

- Generate a set of tree nodes.
- Sort this set in non-increasing order.
- Create a counter that stores a sum of node probabilities.
- Get the first node with maximum self probability.
- Increase the counter.
- Add this node into another set and remove from first.
- Repeat last three operations while counter less then 0.95.
- Build tree from second set of rules.

After performing this operations we get the set of nodes. We could build first tree from second set of rules and second tree from first set of nodes. After this we need to translate the first tree into an assembly language program.

1) Notation used: Let $node1, node2$ — tree vertices, $value$ — some feature value. Let's introduce the following notations:

- $Tree.root$ — the root node of the tree $Tree$.
- $node_1(value)$ — the descendant of the node $node_1$, connected to it by an arc with the mark value.
- $node_1.rules$ — set of rules corresponding to node $node_1$.
- $node_1.edges$ — set of marks of arcs coming from node $node_1$.
- $node_1.prob$ — an amount of probabilities of rules.
- $copy(node_1, val, node_2)$ — a procedure that adds to the node $node_1$ a descendant with an arc marked value, copying the tree that forms the node $node_2$.
- $equals(node_1, node_2)$ — function that returns $true$ if the trees formed by nodes $node_1$ and $node_2$ are the same, otherwise it returns $false$. The comparison takes into account the rule sets and arc labels associated with the nodes.
- $same(rule_1, rule_2)$ — function that returns $true$ if rules are **same**.
- $isleaf(node_1)$ — function that returns $true$ if $node_1$ — tree leaf, otherwise it returns $false$.

2) Flow table optimization algorithm: Let us introduce procedure **Same** (Listing 1), it returns a set of rules that are a union of **same** rules in the sets of two nodes.

Let us introduce flow table optimization algorithm. It can be describe by procedure **Optimize** (Listing 2), where **node** — tree node. For optimizing flow table we need to perform this procedure to $Tree.root$.

```
procedure Same(node_1, node_2):
    rules = {}
    for all rule_1 in node_1.rules do
        for all rule_2 in node_2.rules do
            if same(rule_1, rule_2) then
                rules += {rule_1 union rule_2}
            endif
    return rules
```

Listing 1: Procedure for obtaining a set of rules derivedfrom the same rules

```
procedure Optimize(node):
    if not isleaf(node) then
        for all val_1 in node.edges do
            for all val_2 in node.edges do
                if val_1 not equal val_2 then
                    node.rules += Same(node(val_1
    ), node(val_2))
                endif
        for all val in node.edges do
            Optimize(node(val))
    endif
```

Listing 2: Procedure for optimizing the tree

VI. Evaluation

A. Evaluation methodology

In this section we describe evaluation methodology. Flow table compression algorithms can be evaluated by an assembly language program evaluation. This is so because flow tables translator with implemented data compression algorithms translates flow table into an assembly language program. We used the following parameters in our evaluation:

- An assembly language program memory usage
- An assembly language program average number of instructions requires for one packet processing.

The described analysis requires doing the following actions for each flow table.

1) Choose a flow table for this experiment.
2) Translate the flow table using flow tables translator implementation based on: LPM tree, AVL tree, Our flow table compression algorithm.
3) Execute simulation on the NP simulation model.
4) Evaluate results.

1) Memory usage calculation method: The flow tables translator tool use intermediate flow table representation as trees. Each node of the tree is translated into an assembly language program part. Fully assembled from parts assembly language program has N instructions. Every instruction uses 128 *bits* of memory. Therefore memory usage defined as M can be calculated as:

$$M = 128 * N$$

In our evaluation results we use $KBytes$ to represent memory usage units.

B. Evaluation data

Several variants of the flow tables should be used for the evaluation. These variants cover most usable network protocols. In this section, we will introduce the flow table templates.

- The first pattern — a flow table rule pattern contains the values of three attributes: an input port number, a destination MAC address and a source MAC address.
- The second pattern — a flow table rule pattern contains the values of two attributes: an IPv4 destination address and an IPv4 source address.
- The third pattern — a flow table contains five attributes: an input port number, a destination MAC address, a VLAN ID, a L3-level header ID (EtherType) and a destination IPv4 address.

An example of input data represented in Listing 3.

```
{SRC_MAC , DST_MAC , INSTR}
{SRC_MAC , DST_MAC , INSTR}
+---+-----------+-----------+--------------+
| 1 | :12       | :10:1     | goto_table 1 |
+---+-----------+-----------+--------------+
| 1 | :23:45    | :20       | goto_table 1 |
+---+-----------+-----------+--------------+
| 1 | :0        | :1        | goto_table 1 |
+---+-----------+-----------+--------------+
```

Listing 3: Example of input data for input of a flow table

C. Evaluation results

We performed an evaluation on the simulation model of the NP. This evaluation shows that our solution allowed to use several times less memory of the NP.

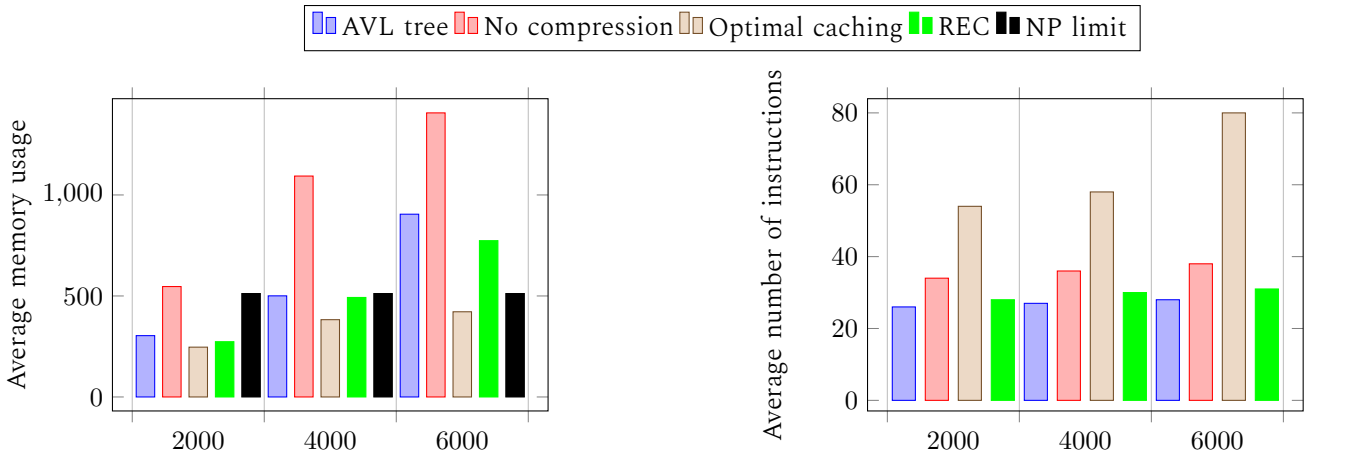
We carried out our research for different sized flow tables. Currently, the maximum size of a flow table is 6000 rules with compression. Flow table without compression can contain only about 1500 rules.

Optimal caching has the best compression rate (Fig. 1a), but the worst average number of instructions required to processing one packet (Fig. 1b). This can be explained by necessity to use many instructions to make the CPU call.

Recursive end-point cutting has less compression rate then optimal caching (Fig. 1a) and better average number of instruction required to processing one packet (Fig. 1b).

VII. Future work

In the future works we will refine evaluation data. We expect less memory usage with our compression algorithm implemented into flow table translator. In the first experiments conducted, we obtained results showing a significant reduction in the amount of memory usage with the help the data compression algorithm. After this we could check possibility of TCAM memory implementation and use this compression algorithms for it.



(a) Network processor memory usage depends different flow table sizes.

(b) Average number of instructions for processing one packet depends flow table size

VIII. Acknowledgements

This work is partially supported by the Russian Foundation for Basic Research under grant 19-07-01076.

References

[1] Smeliansky R.L. "System Defined networks". In: *Open Systems* 9 (2012), pp. 15–26.

[2] Open Networking Foundation. *OpenFlow Switch Specification Version 1.3.0 (Wire Protocol 0x04)*. 2012.

[3] A. Markoborodov, Y. Skobtsova, and D. Volkanov. "Representation of the OpenFlow Switch Flow Table". In: *2020 International Scientific and Technical Conference Modern Computer Network Technologies (MoNeTeC)*. 2020, pp. 1–7. DOI: 10.1109/MoNeTeC49726.2020.9258208.

[4] Wolfgang Braun and Michael Menth. "Wildcard compression of inter-domain routing tables for OpenFlow-based software-defined networking". In: *2014 Third european workshop on software defined networks*. IEEE. 2014, pp. 25–30.

[5] Ori Rottenstreich and János Tapolcai. "Optimal rule caching and lossy compression for longest prefix matching". In: *IEEE/ACM Transactions on Networking* 25.2 (2016), pp. 864–878.

[6] Yeim-Kuan Chang and Han-Chen Chen. "Fast packet classification using recursive endpoint-cutting and bucket compression on FPGA". In: *The Computer Journal* 62.2 (2019), pp. 198–214.